



IdentityIQ Console

Version: 8.3

Revised: April 2022

Copyright and Trademark Notices

Copyright © 2022 SailPoint Technologies, Inc. All Rights Reserved.

All logos, text, content, including underlying HTML code, designs, and graphics used and/or depicted on these written materials or in this Internet website are protected under United States and international copyright and trademark laws and treaties, and may not be used or reproduced without the prior express written permission of SailPoint Technologies, Inc.

"SailPoint," "SailPoint & Design," "SailPoint Technologies & Design," "Identity Cube," "Identity IQ," "IdentityAI," "IdentityNow," "SailPoint Predictive Identity" and "SecurityIQ" are registered trademarks of SailPoint Technologies, Inc. None of the foregoing marks may be used without the prior express written permission of SailPoint Technologies, Inc. All other trademarks shown herein are owned by the respective companies or persons indicated.

SailPoint Technologies, Inc. makes no warranty of any kind with regard to this manual or the information included therein, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. SailPoint Technologies shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Patents Notice. <https://www.sailpoint.com/patents>

Restricted Rights Legend. All rights are reserved. No part of this document may be published, distributed, reproduced, publicly displayed, used to create derivative works, or translated to another language, without the prior written consent of SailPoint Technologies. The information contained in this document is subject to change without notice.

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DOD agencies, and subparagraphs (c)(1) and (c)(2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

Regulatory/Export Compliance. The export and re-export of this software is controlled for export purposes by the U.S. Government. By accepting this software and/or documentation, licensee agrees to comply with all U.S. and foreign export laws and regulations as they relate to software and related documentation. Licensee will not export or re-export outside the United States software or documentation, whether directly or indirectly, to any Prohibited Party and will not cause, approve or otherwise intentionally facilitate others in so doing. A Prohibited Party includes: a party in a U.S. embargoed country or country the United States has named as a supporter of international terrorism; a party involved in proliferation; a party identified by the U.S. Government as a Denied Party; a party named on the U.S. Department of Commerce's Entity List in Supplement No. 4 to 15 C.F.R. § 744; a party prohibited from participation in export or re-export transactions by a U.S. Government General Order; a party listed by the U.S. Government's Office of Foreign Assets Control as ineligible to participate in transactions subject to U.S. jurisdiction; or any party that licensee knows or has reason to know has violated or plans to violate U.S. or foreign export laws or regulations. Licensee shall ensure that each of its software users complies with U.S. and foreign export laws and regulations as they relate to software and related documentation.

Contents

IdentityIQ Console	1
Launching the Console	2
Viewing the List of Commands	3
Command-Line Parameters	7
Piped Commands in the IdentityIQ Console	8
Sample Unix Piped Commands	8
Sample Windows Piped Commands	8
Command Syntax	9
Syntax for Redirecting Command Output	10
IdentityIQ Console Commands	11
Commonly Used Commands	11
Help	16
Index	16
Search	17
Show	20
Less Commonly Used Commands	20
Seldom Used Commands	28

IdentityIQ Console

The IdentityIQ Console is the command line utility for interfacing with IdentityIQ. This section lists the console commands and their descriptions.

Launching the Console

The IdentityIQ Console (iiq console) is launched by running the `iiq.bat` file found in the `installation Directory/WEB-INF/bin` directory. From a command prompt, launch the console with the command as shown for each operating system type:

Windows: `iiq console`

Unix: `./iiq console -j`

The `iiq console` requires the System Administrator capability.

By default, the `iiq console` tries to authenticate with the default user/password `spadmin / admin`. If authentication fails, you are prompted for a user name and password. Specify the user name and password on the command line.

For example:

```
iiq console -u amy.cox -p mypassword
```

The console prompts for user input if the password is omitted, and will not launch if the credentials supplied are not associated with an identity that has console access.

Authentication is disabled if there are no identities. This case is encountered during IdentityIQ setup, before `init.xml` is imported.

The `-j` option turns on the JLine Java library for handling console input, enabling some ease-of-use functions such as command history recall. Command history recall is enabled in Windows without this library, so this parameter is not required in the Windows environment.

The `>` prompt character indicates that the console is running and ready to accept commands.

Viewing the List of Commands

The **help** command displays a list of all commands available in the console along with a short description of each. At the command prompt, enter `help` or `?` to see this full list of available commands.

Command	Description
<code>?</code>	Display command help
<code>help</code>	Display command help
<code>echo</code>	Display a line of text
<code>quit</code>	Quit the shell (same as exit)
<code>exit</code>	Exit the shell (same as quit)
<code>source</code>	Execute a file of commands
<code>properties</code>	Display system properties
<code>time</code>	Show how much time a command takes to run
<code>xtimes</code>	Run a command x times
<code>about</code>	Show application configuration information
<code>threads</code>	Show active threads
<code>logConfig</code>	Reload log4j configuration
Objects	
<code>dtd</code>	Create dtd
<code>summary</code>	Summarize objects
<code>classes</code>	List available classes
<code>list</code>	List objects
<code>count</code>	Count objects
<code>get</code>	View an object
<code>checkout</code>	Checkout an object to a file
<code>checkin</code>	Checkin an object from a file
<code>delete</code>	Delete an object
<code>rollback</code>	Rollback to a previous version
<code>rename</code>	Rename an object
<code>import</code>	Import objects from a file

Command	Description
importManagedAttributes	Import managed attribute definitions from a CSV file
export	Export objects to a file
exportManagedAttributes	Export managed attribute definitions to a CSV file
exportJasper	Exports only the jasperReport xml contained in a JasperTemplate object
associations	Show target associations for an object
Identities	
identities	List identities
snapshot	Create an identity snapshot
score	Refresh compliance scores
listLocks	List all class locks
breakLocks	Break all class locks
Tasks	
tasks	Display scheduled tasks
run	Launch a background task
runTaskWithArguments	Launch a task synchronously with arguments
terminate	Terminate a background task
terminateOrphans	Detect and terminate orphaned tasks
restart	Restart a failed task if possible
send command	Send an out-of-band task command
taskProfile	Display task profiling report
Certifications	
certify	Generate an access certification report
cancelCertify	Cancel an access certification report
archiveCertification	Archive and delete an access certification report
decompressCertification	Decompress an access certification archive
Groups	
refreshFactories	Refresh group factories (but not groups)
refreshGroups	Refresh groups (but not factories)
showGroup	Show identities in a group

Command	Description
Workflow	
workflow	Start a generic workflow
validate	Validate workflow definition
workItem	Describe a work item
approve	Approve a work item
reject	Reject a work item
wftest	Run the workflow test harness
Tests	
rule	Run a rule
parse	Parse an XML file
warp	Parse an XML object and print the re-serialization
notify	Send an email
authenticate	Test authentication
authenticateWithOptions	Test authentication with options
simulateHistory	Simulate trend history
search	Run a simple query
textsearch	Run a full text search
certificationPhase	Transition a certification into a new phase
impact	Perform impact analysis
event	Schedule an identity event
expire	Immediately expire a workitem that has an expiration configured. If the workitem is type Event it'll also push the event forward with the workflow
connectorDebug	Call one of the exposed connector methods using the specified application
encrypt	Encrypt a string.
sql	Execute a SQL statement
hql	Execute a search based on a Hibernate Query Language statement.
updateHql	Update the hql search.
date	Displays the current system date/time and its UTIME (universal

Command	Description
	time) value (Optional UTIME parameter causes the command to display the date/time corresponding to the provided UTIME value.)
shell	Escapes out to the command line and run the command specified.
meter	Toggles metering on and off; while metering is on, the console reports some timing statistics for each command executed. Meter information is displayed after the results of each command as it is executed.
compress	Compress the contents of a file to a string that can be included within an XML element.
uncompress	Return a compressed, Base64-encoded file to its uncompressed format.
clearEmailQueue	Remove any queued emails that have not been sent
provision	Evaluate a provisioning plan
lock	Lock an object
unlock	Break a lock on an object
showLock	Show lock details
clearCache	Clear the object cache
service	Service management
oconfig	Analyze ObjectConfigs
plugin	Install and manage plugins
recommender	Manage and test recommendations

Command-Line Parameters

You can use parameters with the `iiq` console commands to manage command behavior.

Command	Description
<code>-u <username></code> <code>-p <password></code>	Run the console using the supplied username and password for authentication. Example: <code>-u mary.johnson -p mypwd</code> If you provide a username but omit the password, the <code>iiq</code> console prompts for the password value.
<code>-j</code>	Used in Unix systems only. Adds improved command editing and history support. Use <code>cursor-up</code> and <code>cursor-down</code> to navigate console command history.
<code>-h <hostname></code>	Override the hostname used for the console. Example: <code>-h consoleA</code>
<code>-c <command></code>	Run the given command, and then exit. Example: <code>-c "list Certification"</code>
<code>-f <filename></code>	Run the commands read from the provided file, then exit. Example: <code>-f myCommands.txt</code>
<code>-e <CSV of services to start></code>	Automatically start the services specified in the provided comma-separated list. Example: <code>-e Heartbeat,Task,Reanimator</code>
<code>-heartbeat</code>	Force the Heartbeat service to automatically start. This is the equivalent to <code>-e Heartbeat</code>

Piped Commands in the IdentityIQ Console

The iiq console commands can use piping as a useful way to filter or redirect output.

You can use standard Unix or Windows commands to process the output of your iiq console commands. You must use the commands (Unix or Windows) that are appropriate to your own operating system. Unix commands will not work in a Windows environment, and vice versa.

Sample Unix Piped Commands

List only manual workitems:

```
list workitem | grep manual
```

Get a count of all your policies:

```
list policy | wc -l
```

List policies, excluding any SOD (separation of duties) policies, sort them in reverse order, and write the output to a text file:

```
list policy | grep -v SOD | sort -r | > MyNonSODPoliciesInReverseOrder.txt
```

Sample Windows Piped Commands

List only manual workitems:

```
list workitem | findstr manual
```

List policies and direct the list output to the clipboard:

```
list policy | clip
```

List all the "Example" rules in your system, and write the output to a text file:

```
list rule | findstr Example | > AllExampleRules.txt
```

Command Syntax

The syntax for any console command that requires parameters can be determined by entering that command with no arguments.

```
> workflow
```

```
usage: workflow name [varfile]
```

Command names are case sensitive and must be entered as shown in the command list. Parameters are not case sensitive.

Some commands take no arguments and execute if entered. This table contains a list of the commands that require no arguments.

Command	Action
? or help	Lists all available console commands
quit or exit	Exits the console shell
classes	Lists all classes
refreshGroups	Refresh group indexes (Optional group name or ID can be specified)
refreshFactories	Refresh set of GroupDefinitions for a GroupFactory (Optional factory name or ID can be specified)
logConfig	Reloads log4j configuration from log4j2.properties file
summary	Lists all classes and the count of objects of that class in the system
properties	Displays Java properties of the server where IdentityIQ is installed
about	Displays application configuration information
threads	Shows a list of active threads
tasks	Writes a list of all currently scheduled tasks, in a columnar layout, to the console (stdout)
identities	Writes the Name, Manager, Roles, and Links for each Identity in the system to the console (stdout)
date	Displays the current system date/time and its UTIME (universal time) value (Optional UTIME parameter causes the command to display the date/time corresponding to the provided UTIME value.)
status	Reports current running status of the task and request schedulers
meter	Toggles metering on and off; while metering is on, the console reports some timing statistics for each command executed. Meter information is displayed after the results of each command as it is executed.
clearEmailQueue	Deletes all queued but unsent email messages

Command	Action
clearCache	Clears the IdentityIQ object cache

Syntax for Redirecting Command Output

Most of the commands report data or error messages to the console or standard out (stdout) for the system. The output for any command can be redirected to a file by specifying `> filename` at the end of the command.

This example redirects the output from the **get** command to a file:

```
> get identity Adam.Kennedy > c:\output\AdamKennedyID.xml
```

IdentityIQ Console Commands

In this section, the list of console commands is subdivided based on how frequently they are likely to be used in a production environment.

Commonly Used Commands

This section lists and documents the syntax and actions of the most commonly-used console commands.

Help and ?

These two commands list all the available console commands.

Syntax	? help
Examples	> ? > help
Result	Lists all commands available in the console

Exit and Quit

These two commands exit the console shell, returning the user to the operating system command prompt.

Syntax	exit quit
Examples	> exit > quit
Result	Exits console shell and returns user to the operating system command prompt

Source

The **source** command runs commands from a script file. The commands on each line in the file are executed by the console sequentially.

Syntax	source <i>filename</i>
Examples	source c:\data\cmdfile.txt
Result	Runs the console commands in the c:\data\cmdfile.txt file sequentially

List

The **list** command lists all objects of the specified class, constrained by any specified filter. If this command is specified without arguments, the command syntax is displayed, followed by a list of all available classes whose objects can be listed. This is helpful in locating objects within the system and in identifying object names to use as parameters on other commands.

Syntax	<pre>list <i>classname</i> [<i>filter</i>] filter: xxx - names beginning with xxx xxx* - names beginning with xxx *xxx - names ending with xxx *xxx* - names containing xxx</pre>
Examples	> list application ent*
Result	Lists all application objects whose names begin with ent

Get

The **get** command displays the XML representation of the named object.

Syntax	get <i>classname</i> < <i>objectname</i> or <i>ID</i> >
Examples	> get identity Adam.Kennedy
Result	Displays the Adam.Kennedy Identity in XML format

This command only displays the object to the console (stdout), it does not export the object. The output can be redirected to a file if the user has write access to the server's file system.

```
> get identity Adam.Kennedy > c:\output\AdamKennedyID.xml
```

Other alternatives for getting the XML representation of an object into a text file include:

- copying and pasting contents of this command's stdout into a text file
- retrieving the object's XML from the IdentityIQ Debug pages
- using the **checkout** command (described next) to write the XML representation of an object to a text file

Checkout

The **checkout** command writes a copy of the XML representation of the requested object to the specified filename. The file can be used for review or for moving objects from one environment to another, for example, from the user acceptance testing environment to production. Organizations doing custom development on rules, workflows, etc. might use **checkout** to extract any of these objects to a file for modification.

Syntax	checkout <i>class name</i> < <i>objectname</i> or <i>ID</i> > <i>file</i> [-clean [=id,created...]]
Examples	> checkout rule "Cert Signoff Approver" certrule.xml

Result	Writes a copy of the Cert Signoff Approver rule's XML representation to the file certrule.xml
--------	---

The **-clean** option can be used to remove all values that do not transfer between IdentityIQ instances, such as created and modified dates as well as globally unique ID values (GUIDs). Specifying the **-clean** option with no qualifiers cleans the id, created, modified, and lastRefresh attributes. The **-clean** option can also be used to explicitly clear specific fields by name. The fields to clear must be listed in a comma separated list.

Checkin

The **checkin** command reads a file containing an object's XML representation and saves the object into the database. If the object is a workItem, the command invokes the workflower to process the workItem. If the object is a bundle (role) and the approve parameter is specified, a role approval workflow is launched. For all other object types, and for bundles that are submitted without the approve parameter, the object is saved into the database.

The command's syntax parsing allows the approve parameter to be specified for any object but it only impacts the processing on Bundle objects.

Syntax	checkin <i>filename</i> [approve]
Examples	> checkin newRole.xml approve > checkin bobSmithID.xml
Result	First example saves Identity Bob Smith, as represented by the XML in bobSmithID.xml, into to the database; overwrites existing or adds new record Second example launches an approval workflow for the bundle object represented by the XML in newRole.xml

If an Import file is specified as the input file for this command, only the first object in the file is checked in; the rest are ignored and a warning message is displayed to the console (stdout).

Delete

This action cannot be undone and should be used with extreme caution and only in rare circumstances.

The **delete** command deletes the named object and removes all of its owned, or subordinate, objects. In a production environment, this is not recommended unless specifically directed by IdentityIQ Support.

Syntax	delete <i>classname</i> < <i>objectname</i> or <i>ID</i> >
Examples	> delete identity bob.smith
Result	Removes Identity Bob Smith and all of his associated objects from the system

Wildcards can be used on the <*object name* or *ID*> argument:

- * - all objects of the specified class (use with extreme caution!)
- xxx - all objects whose name or ID contains xxx

Import

The **import** command imports objects into IdentityIQ from an XML file. This command can be used on a file that contains a Jasper report, a IdentityIQ import file, or an object of one of the standard object classes. The file contents are evaluated and processed based on the first tag in the file:

- *JasperReport*: Jasper report
- *IdentityIQ*: IdentityIQ import object; can contain multiple regular objects in one file as well as an ImportAction tag that directs how the contents of the file are processed, for example, merge, include, execute, logConfig.
- Anything else: assumed to be a single regular object

Syntax	<code>import [-noids] filename</code>
Examples	<pre>> import init.xml > import -noids init.xml</pre>
Result	<p>The first example Imports the contents of the file init.xml into the IdentityIQ data-base.</p> <p>In the second example, all ID attributes are removed before parsing occurs.</p> <p>This action is a normal part of the initialization process for IdentityIQ.</p>

Syntax	<code>import [-noids][-noroleevents] filename</code>
Examples	<pre>> import -noids bundles.xml > import -noids -noroleevents bundles.xml</pre>
Result	<p>The first example allows user to import the events. It removes all ID attributes before parsing.</p> <p>The second example disables generation of role change events for role propagation.</p> <p>Select 'Allow Role Propagation' option from the Global Settings -> IdentityIQ Configuration -> Roles option in UI.</p>

This is one of the most commonly used commands. Installations who manage their workflows and rules in an external source code control system, for example, use this command to bring changes to those objects into IdentityIQ once they have been modified in their external XML representations.

Export

The **export** command writes all objects of a given class to a specified filename. This is commonly used in gathering objects from IdentityIQ to deliver to IdentityIQ Support as resources in resolving tickets. It is also used for moving sets of objects between environments and for managing objects outside of IdentityIQ, such as storing workflows and rules in a source code control system.

More than one class can be exported at a time to the same file by specifying all the desired class names as arguments to the command. If the **export** command is specified without any class names, all objects of all classes are exported to the specified filename.

Syntax	<code>export [-clean[=id,created...]] filename [classnameclassname ...]</code>
Examples	<pre>> export -clean workflows.xml workflow > export IdLink.xml identity link</pre>
Result	<p>The first example exports the entire set of workflow objects from IdentityIQ to the file <code>workflows.xml</code>, removing values from the <code>id</code>, <code>created</code>, <code>modified</code>, and <code>lastRefresh</code> attributes.</p> <p>The second example exports all identities and links to a single file.</p>

ListLocks

The **listLocks** command lists all locks held on any objects of the named class. At this time, Identity is the only class for which this command operates.

BreakLocks

This command should be used with caution. Locks are useful in maintaining data integrity, and breaking them at the wrong time can potentially permit conflicting updates that can result in data corruption.

The `unlock` command can be used to break a single lock whereas this command breaks all locks held on any object in the specified class.

If a process is holding a lock but is unable to perform the required action, the lock can cause problems in other processes' performance as well. The **breakLocks** command can be used to release locks forcibly. At this time, Identity is the only class for which this command functions.

Syntax	<code>breakLocks classname</code>
Examples	<pre>> breakLocks identity</pre>
Result	Releases all locks held on any identity object in the system and reports to the console the identity name, lock holder, and UTIME value for the lock date/time and the lock expiration date/time.

Rule

The **rule** command runs a rule defined in the system. The rule to run is specified as a command parameter. If any input variables must be passed to the rule, they must be entered in a variable file, specified as an XML Map. The file name is then also passed as a parameter to the command.

This command can be used for testing or executing existing system rules. It can also be used to run any BeanShell code snippet against the IdentityIQ database. The code is created as a rule and loaded into the system and then executed from the console. Support uses rules like this for data cleanup.

Syntax	<code>rule <rulename or ID> [varfile]</code>
--------	--

Examples	> rule "Check Password Policy" c:\data\pwdParams.xml
Result	Runs the Check Password Policy rule, passing its input variables through the file c:\data\pwdParams.xml

Parse

The **parse** command validates an XML file. If it is in valid form and its tags match the IdentityIQ DTD, it runs successfully and no information is printed to the console (stdout). If errors are encountered, a `RuntimeException` is printed to the console describing the error.

Syntax	<code>parse filename</code>
Examples	> parse c:\data\newWorkflow.xml
Result	Validates the XML in the file c:\data\newWorkflow.xml and reports any errors to the console

Rolerelationship

The **rolerelationship** command lets you view, search, and rebuild relationships between roles and their entitlement profiles, required roles, inherited roles, and permitted roles.

Syntax

This command requires a subcommand to specify the particular action you want the command to take.

Details about subcommands and their options are provided below; the general syntax for running the command with a subcommand is:

```
rolerelationship subcommand
```

Help

This subcommand returns help details about other subcommands and their usage details.

Syntax: `rolerelationship help`

Index

This subcommand builds Bundle Profile Relation records, based on the provided role name, ID or type. Bundle Profile Relation records provide a snapshot-level view of a role and its relationships to entitlements, required roles, inherited roles, permitted roles, and managed attributes. This command runs the Refresh Role-Entitlement Associations task, with the filters that are specified in the command syntax applied.

Note that the only function currently supported is the build function, which must be included in the command syntax.

Syntax: `rolerelationship index [function] [filter]`

Two filter options can be used with the index subcommand. Wildcards are not supported in these filters.

- `role_name`: The role name or ID. Role names that include a space must be enclosed in single quotes (for example, 'Payroll Approver').
- `role_type`: The role type, such as **business** or **it**

Examples

```
rolerelationship index build -role_name 'Payroll Approver'
```

```
rolerelationship index build -role_type it
```

```
rolerelationship index build -role_name 'Tax Manager' -role_type business
```

Search

This subcommand searches for roles using search filters and parameters. You can use a variety of filters and fixed-value parameters to search for roles and role relationships. The `-l` option returns results in a list view, suppressing entitlement and status information in the results, so that relationship data is not displayed.

Syntax: `rolerelationship search [filter] [-l]`

Filters for the Search Subcommand

Use filters to search for roles based on a variety of criteria. Filters can be combined with fixed values (which are described in a later table) to refine your searches.

applications

Syntax is `-app <app_name_or_id> or -app_status <app_status>`.

`- app <app_name_or_id>` returns all roles that have entitlements from the provided application.

`-app_status <app_status>` returns all roles that have entitlements from the provided application with this type of status.

roles

Syntax is `-role_name <role_name_or_id> or -role_type <role_type>`

`-role_name <role_name_or_id>` returns roles matching the provided role name or id.

`-role_type <role_type>` returns roles of this type. Options are **it**, **business**, **assignOrDetect**, **rapidSetupBirthright**, or any custom role types that are requestable.

Roles supports wildcard search using %

entitlements

Syntax is `-ent_attr <attribute> or -ent_value <value>`

`-ent_attr <attribute>` filters by type of profile = entitlement. Options are **group**, **groupmbr**, and **memberOf**.

`-ent_value <value>` filters by type of profile = entitlement and returns only entitlements that contain this value.

Entitlements supports wildcard search using %

permissions

Syntax is `-perm_right <right> or -perm_target <target>`

`-perm_right <right>` filters by type of profile = permission. Options are **create**, **delete**, **execute**, **read**, and **update**.

`-perm_target <target>` filter by type of profile = permission (options are targets defined on entitlement permissions).

Permissions supports wildcard search using %

relationships

Syntax is `-inheritance <inheritance>` or `-rel2role <relationship_to_role>`

`-inheritance <inheritance>` filters by relationship and inheritance. Options are **SELF_ONLY** which defines privileges derived from role only (directly), or **INHERITED_ONLY** which defines privileges derived only through inheritance.

`-rel2role <relationship_to_role>` filters by relationship of the profile to the role. Options are:

- **ANY** ignores all relationships and displays all
- **SELF_ONLY** only shows roles that have the entitlements/permissions directly on them
- **SELF_AND_SPECIFIC_RELATED** shows roles that have the entitlements/permissions directly on it OR a specific relationship (inherited, permitted, required).
- **ALL_RELATED_ONLY** only shows roles that have the entitlement/permission through a specific relationship (relationship cannot be selected in additional filters), not on the role directly, only via derived roles.
- **SPECIFIC_RELATED_ONLY** only shows roles that have the entitlement/permission through a specific relationship (relationship can be selected in additional filters), not on the role itself.
- The `-rel2role` options [**ANY** | **SELF_ONLY** | **ALL_RELATED_ONLY**] are incompatible with specified [`inheritance` | `req_perm_type`] filters other than **ANY**.

relationship types

Syntax is `-ent_perm_type <relation_type>`

Filter by type of entitlement within role. Options are **entitlement** or **permission**.

Value of **permission** is incompatible with `ent_attr` and `ent_value` criteria.

Value of **entitlement** is incompatible with `perm_right` and `perm_target` criteria.

required or permission

Syntax is `-req_perm_type <req_perm_type>`

Filter by **required** or **permitted**. This filter is only applicable when the `-rel2role` fixed value is set to **SELF_AND_SPECIFIC_RELATED** or **SPECIFIC_RELATED_ONLY**

The options for `-req_perm_type` are:

- **ANY** - required or permitted role
- **REQUIRED_ONLY** - required only
- **PERMITTED_ONLY** - permitted only
- **NEITHER** - neither required nor permitted

Fixed Values for the Search Subcommand

Fixed value options in the Search subcommand let you search for roles using specific values. Fixed values can be combined with filters to refine your searches.

<rel2role> (relationship to role)

- **ANY**
- **DIRECT_ONLY** - Only show where privilege is derived directly from the role
- **DIRECT_SELECTED_INDIRECT** Show direct relationships, and any specified indirect relationships
- **ALL_INDIRECT** - Only show where privilege is derived indirectly from other roles
- **SELECTED_INDIRECT** - Only show where privilege is derived indirectly from selected other roles, using entitlement/permission filters

<relation_type>

- **ANY**
- **ENTITLEMENT** - Entitlements only
- **PERMISSION** - Permissions only

<inheritance>

- **ANY**
- **SELF_ONLY** - Privileges derived from the role only
- **INHERITED_ONLY** - Privilege derived from inheritance only

<app_status>

- **APP_NOTFOUND** - Application is not found, role is orphaned
- **ENT_NOTFOUND** - Entitlement not found
- **OK** - Application exists

<req_perm_type>

- **ANY**
- **REQUIRED_ONLY** – Only Required is set to true
- **PERMITTED_ONLY** – Only Permitted is set to true
- **NEITHER** - Neither Required nor Permitted are set to true
- **BOTH** - Both Required and Permitted are set to true

Notes on the Search Subcommand

- The <rel2role> (relationship_to_role) options [**ANY** | **DIRECT_ONLY** | **ALL_INDIRECT**] are incompatible with specified [*inheritance* | *req_perm_type*] filters other than **ANY**
- *ent_perm_type* value of permission is incompatible with *ent_attr* and *ent_value* criteria

- `ent_perm_type` value of entitlement is incompatible with `perm_right` and `perm_target` criteria
- `role_name`, `ent_attr`, `ent_value`, `perm_right`, and `perm_target` support wildcard search with %

Examples

```
rolerelationship search
```

Returns usage/help information for the Search subcommand

```
rolerelationship search -app ExampleRoleApp
```

Returns all roles that have entitlements from the ExampleRoleApp application, indirectly or directly

```
rolerelationship search -app ExampleRoleApp -role_name 'MyCompany IT Role'
```

Returns all direct entitlements in the ExampleRoleApp for the MyCompany IT Role

```
rolerelationship search -app ExampleRoleApp -role_type business
```

Returns all roles of type business that have entitlements in the ExampleRoleApp application, indirectly or directly

```
rolerelationship search -app Oracle_DB_oasis -ent_perm_type permission -perm_target AP_Logins -perm_right create
```

Returns roles in the Oracle_DB_oasis app that have create permissions on the AP_Logins target

Show

This subcommand returns details about entitlements and status for the requested role. A role name or ID must be provided as an argument. If there are any spaces in the role name, the role name must be enclosed in single quotes (for example, 'Payroll Approver'). Wildcards are not supported for the show subcommand.

Syntax: `rolerelationship show [Role Name or ID] [-l]`

The `-l` option returns results in a list view, suppressing entitlement and status information in the results, so that relationship data is not displayed.

Examples

```
rolerelationship show -role_name 'Payroll Approver'
```

```
rolerelationship show -role_name TaxManager -l
```

```
rolerelationship show -role_name 7f0000017e731ebe817e73cf168a0398
```

Less Commonly Used Commands

These commands are not frequently used in a production environment. However, it is helpful to understand and be able to use them when the need arises.

DTD

The DTD command writes the IdentityIQ DTD (Document Type Definition) to the specified file.

Syntax	<code>dtd filename</code>
--------	---------------------------

Examples	> dtd c:\DTD\IdentityIQ.dtd
Result	Writes the IdentityIQ DTD to the file c:\DTD\IdentityIQ.dtd

Classes

The **classes** command lists all classes accessible from the console. These are frequently used as parameters to other commands so this list can be helpful in entering correct arguments on those commands.

Syntax	classes
Examples	> classes
Result	Lists class names for all classes accessible to the console

Count

The **count** command returns a count of the objects of the specified class.

Syntax	count <i>classname</i>
Examples	> count identity
Result	Displays the count of Identity objects in the system

ImportManagedAttributes

The **importManagedAttributes** command is used to set managed attribute values, including localized descriptions, through a CSV file import. This can be used to update existing managedAttributes or to create new ones.

The filename can be specified with an absolute path or can be specified relative to the current working directory. These are the specific requirements for the import file contents:

- The first line in the file must be a comment line (starting with #) that contains the column names for the data records. Column names must be specified in a comma-separated format. All column names must match managedAttribute standard or extended attributes or specify a locale/supported language.
- Subsequent comment lines can be used to specify default values for attributes that are not contained in the data records. For example, if the whole file relates to a single application, the application name could be set as a default through a single comment line.
- Blank lines are permitted in the file, and are ignored, but cannot precede the first comment line.
- The data records must consist of comma-separated data values.

Only types Entitlement and Permission are valid. Group managedAttributes are stored in IdentityIQ as a subcategory of Entitlement type managedAttributes.

- Required attributes are type, application, attribute, and value. If type is not specified in the file, type Entitlement is assumed. The others three properties must be specified in the file. Type, application, and attribute can be specified through the data columns or with a single default value in extra comment lines. The value attribute must be in the data columns and cannot have a default value.
- The data values in the columns named to match supported languages should contain the description to use for that locale.

Example File Contents:


```
# value, displayName, en_US, owner
#   owner=Jeff.Wilson
# application=AD
# attribute=MemberOf
# type=Entitlement
# "CN=administrators,CN=Roles,DC=iiq,DC=com", Admins, "Administrators group",
# "CN=VPN,CN=Roles,DC=iiq,DC=com", VPN, "Remote Workers", Bob.Smith
```

The `test` option causes the command to parse and validate the file without saving changes to the database.

Syntax	<code>importManagedAttributes filename [test]</code>
Examples	<code>> importManagedAttributes "c:\data\managedattributes.csv"</code>
Result	Imports managed attribute data from the file <code>c:\data\managedattributes.csv</code> , updating existing attributes or creating new ones from the data

ExportManagedAttributes

The **exportManagedAttributes** command exports either object properties or descriptions for managedAttributes to a CSV file. This is used to make mass changes to the managed attributes definitions or for collecting all the managed attributes in one file to review as a group.

Syntax	<code>exportManagedAttributes filename [application] [language]</code>
Examples	<code>> exportManagedAttributes "c:\data\AdamManAttrDesc.csv" ADAM en_US</code>
Result	Exports the managed attribute description on ADAM application to the file <code>c:\data\AdamManAttrDesc.csv</code> .

Application and language are both optional arguments and can be specified in either order. At most one application and one language can be specified at a time. If no application name is specified, managed attributes for all applications are exported. If a language is specified, only the core identifying properties of the managed attributes (type, application, attribute, value) and the descriptions for the specified locale are exported. If a language is not specified, all other object properties except descriptions are exported.

The file format generated by this command can be used in the **importManagedAttribute** command, so this command can be used to write values to a file for editing and reimporting. When an application name is specified on the export command, the application is not shown in the data rows but is specified as a default in the file header comments, as described and illustrated in the command details.

Run

The **run** command starts execution of a task that requires and accepts no arguments. Three optional parameters can be specified for this command: `trace`, `profile`, and `sync`.

- **Trace** — writes to the console (stdout) a trace of what happens as the task is run, depending on how the task's tracing code is written.
- **Profile** — displays the timing of certain phases of the task, the details displayed depend on the task's profile code.

- **Sync** — runs the task in synchronous execution mode, as opposed to scheduling it to run in background. If sync is specified, the control returns to the console only after the task has completed and any error messages are written to the console. If sync is not specified, the task is launched in the background and the results of the task are viewable in the taskResults object and are accessible from the console or from the user interface under **Setup -> Tasks -> Task Results**.

Syntax	run <i>taskname</i> [trace] [profile] [sync]
Examples	> run "Refresh Risk Scores"
Result	Runs the Refresh Risk Scores task in background

RunTaskWithArguments

The **runTaskWithArguments** command starts execution of a task that requires arguments. These tasks are always run in synchronous execution mode. This can only be used for tasks that accept arguments of simple data types; specifying an object as an argument is not possible here.

Syntax	runTaskWithArguments <i>taskname</i> [arg1=val1,arg2=val2,...]
Examples	> runTaskWithArguments "Identity Refresh" refreshLinks=True,promoteAttributes=False
Result	Runs the Identity Refresh task, refreshing Links for the Identities

Restart

The **restart** command restarts execution of a task that failed.

Syntax	restart <i>taskResultName</i>
Examples	> run "Refresh Risk Scores"
Result	Restarts the Refresh Risk Scores task in background if possible

RefreshFactories

The **refreshFactories** command can be specified with no arguments to refresh all group factories or with a specific GroupFactory name. Refreshing the group factory means identifying the group values that define each of the groups (or GroupDefinition objects). It does not refresh the list of Identities that make up each group or the statistics gathered for each group - just the list of groups themselves.

Syntax	refreshFactories [< <i>factorname</i> or <i>ID</i> >]
Examples	> refreshFactories
Result	Refreshes the GroupDefinition list associated with each GroupFactory in the system

RefreshGroups

The **refreshGroups** command refreshes the group indexes for all groups or for the specified group named as a command argument. The group indexes are collections of statistics for identities that are part of the group. The statistics

include number of members, number of certifications due for certification owners in the group, number of certifications owned and completed on time by members of the group, and risk score information for members of the group. RefreshGroups only applies to GroupDefinitions that are indexed (attribute indexed="True").

Syntax	refreshGroups [<i>groupname</i> or <i>ID</i>]
Examples	> refreshGroups
Result	Refreshes index information for all indexed groups

ShowGroup

The **showGroup** command shows the membership (Identities) of the group named as an argument to the command.

Syntax	showGroup < <i>groupname</i> or <i>ID</i> >
Examples	> showGroup Finance
Result	Lists all Identities who are members of the Finance group.

Workflow

The **workflow** command launches the workflow specified as a command parameter. Input variables must be entered in a variable file to get passed to the workflow. A variable file is specified as an XML Map. The file name is then also passed as a parameter to the command. When the workflow is successfully launched, the XML for the workflowCase is printed to the console (stdout).

Syntax	workflow < <i>workflowname</i> or <i>ID</i> > [<i>varfile</i>]
Examples	> workflow "LCM Provisioning" c:\data\provFile.xml
Result	Runs the LCM Provisioning workflow, passing its input variables through the file c:\data\provFile.xml

The varfile should contain an attributes map like the example shown below. This example map passes an identity name and a provisioning plan object to the workflow.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Attributes PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<Attributes>
  <Map>
    <entry key="identityName" value="Adam.Kennedy"/>
    <entry key="plan">
      <value>
        <ProvisioningPlan>
          <AccountRequest application="IIQ" nativeIdentity="Adam.Kennedy" op="Modify">
            <AttributeRequest name="assignedRoles" op="Add" value="PRISM User">
              <Attributes>
                <Map>
                  <entry key="comments" value="req A"/>
                </Map>
              </Attributes>
            </AttributeRequest>
          </AccountRequest>
        </ProvisioningPlan>
      </value>
    </entry>
  </Map>
</Attributes>
```

```

    </AccountRequest>
    <AccountRequest application="IIQ" nativeIdentity="ABC_12345" op="Modify">
      <AttributeRequest name="assignedRoles" op="Add" value="Test Role B2">
        <Attributes>
          <Map>
            <entry key="comments" value="req B"/>
          </Map>
        </Attributes>
      </AttributeRequest>
    </AccountRequest>
  </ProvisioningPlan>
</value>
</entry>
</Map>
</Attributes>

```

Validate

The **validate** command can validate a workflow or a rule. Input variables must be entered in the variable file to be passed to a workflow or rule. The variable file is specified as an XML map and the file name is passed as a parameter to the command. Validation errors are printed to the console (stdout).

Syntax	<code>validate <rule or workflow name or ID> [varfile]</code>
Examples	<code>> validate "LCM Provisioning" c:\data\provFile.xml</code>
Result	Validates the LCM Provisioning workflow, passes the input variables through c:\data\provFile.xml, and displays any validation errors

See [Workflow](#) for an example of the varfile format.

Wftest

The **wftest** command is used to one or more workflows. The *WorkflowTestSuite* can be the name of a WorkflowTestSuite object or a file containing one.

Syntax	<code>wftest WorkflowTestSuite name filename</code>
Examples	<code>> wftest c:\test\workflowTest.xml name c:\test\workflowTestOut.xml</code>
Result	Tests the workflows and sends the outcome to the workflowTestOut file.

SQL

The **sql** command executes a SQL statement. It can execute SQL specified inline with the command or it can read the SQL from a file. Only one SQL statement can be executed at a time. The output can be printed to the console (stdout) or redirected to a file. Select, update, and delete SQL statements can be executed. Update and delete actions cannot be undone.

Syntax	<code>sql sqlStatement -f inputFileNames</code>
Examples	<code>> sql "select * from sptr_identity" > c:\data\Identities.dat</code> <code>> sql -f c:\sql\SelectIdentities.sql</code>

Result	The first example executes the specified select statement and writes the results to c:\data\Identities.dat. The second example reads the SQL from c:\sql>SelectIdentities.sql, prints the SQL to the console (stdout), and displays the query results to the console (stdout).
--------	---

Provision

The **provision** command processes the specified provisioning plan for the specified identity but does not save the information. This is used to test a connector or to test a provisioning plan. Errors are reported to the console. If the provisioning action would succeed, nothing is reported to the console.

Syntax	Provision <i><identityname or ID> provisioningPlanFilename</i>
Examples	> provision Adam.Kennedy c:\data\provFile.xml
Result	Tests the provisioning plan contained in c:\data\provFile.xml against Identity Adam.Kennedy and reports any exceptions to the console

The provisioning plan file should contain a provisioning plan in XML format. For example:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE ProvisioningPlan PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<ProvisioningPlan>
  <AccountRequest application="IIQ" nativeIdentity="ABC_12345" op="Modify">
    <AttributeRequest name="assignedRoles" op="Add" value="Test Role B1">
      <Attributes>
        <Map>
          <entry key="comments" value="req A"/>
        </Map>
      </Attributes>
    </AttributeRequest>
  </AccountRequest>
  <AccountRequest application="IIQ" nativeIdentity="ABC_12345" op="Modify">
    <AttributeRequest name="assignedRoles" op="Add" value="Test Role B2">
      <Attributes>
        <Map>
          <entry key="comments" value="req B"/>
        </Map>
      </Attributes>
    </AttributeRequest>
  </AccountRequest>
</ProvisioningPlan>
```

Lock

The **lock** command obtains a persistence lock on an object. The object's class and ID or name must be specified. By default, the lock is issued to the username Console, but a different username can be specified in the command's lock-Name parameter. The lock automatically expires after 5 minutes.

Syntax	lock <i>classname <objectID or name> [lockName]</i>
Examples	> lock identity John.Smith

Result	Obtains a persistence lock for Console on the identity record for John.Smith
--------	--

Identity objects are the only objects that can be locked. Attempts to specify a different object type in this command results in a syntax error exception.

Unlock

The **unlock** command releases the lock on an object. The object's class and ID or name must be specified. If the object is not locked, the message "Object is not locked" is displayed. If it is locked, the lock is released and the message "Lock has been broken" is displayed.

Syntax	<code>unlock <i>classname</i> <objectID or name></code>
Examples	<code>> unlock identity John.Smith</code>
Result	Breaks the lock on the identity record for John.Smith

ShowLock

The **showLock** command lists the lock owner, locked date/time, and lock expiration date/time for a locked object. The object's class and ID or name must be specified to view its lock information. The message Object is not locked is displayed if the object is not currently locked. If the lock has expired, the lock information is shown but is prefaced with the message Lock has expired.

Syntax	<code>showLock <i>classname</i> <objectID or name></code>
Examples	<code>> showLock identity John.Smith</code>
Result	Displays lock information (owner, date/time, expiration date/time) on the identity record for John.Smith or displays Object is not locked

Oconfig

The **oconfig** command list all extended attributes defined for each class that supports extended attributes. The list indicates the extended attribute numbers and corresponding attribute names on each class. Identity extended attributes which link to other Identities are stored separately in extended identity attribute fields, so those are listed in a separate Extended Identity Attributes sub-list under the Identity ObjectConfig. The objectConfig detail displays No attributes defined if no extended attributes are defined for a given class. An Object not found message is displayed if no objectConfig exists for the class.

Syntax	<code>oconfig</code>
Examples	<code>> oconfig</code>
Result	Displays each objectConfig and its extended attributes, numbered according to the extended attribute number that corresponds to each ObjectConfig: Identity 1 region 2 Department 3 location 4 empld 5 jobtitle

```

Extended Identity Attributes:
1 regionOwner
2 locationOwner
ObjectConfig: Link
1 inactive
2 service
3 privileged
4 lastLogin
ObjectConfig: Application
1 DeployDate
ObjectConfig: Bundle
No attributes defined
ObjectConfig: ManagedAttribute
1 authorization
2 email
3 rank
ObjectConfig: CertificationItem
Object not found
    
```

TextSearch

The `textsearch` command enables command-line execution of full text searches as they are done through the LCM full text search. The class name must be either `ManagedAttribute` or `Bundle`, since those are the only indexed classes. This command searches for the specified string in the `fullTextIndex` created for the specified class and returns a map representation of the objects in which the string is found. If a filter attribute and value are specified, the search is further constrained to entries that correspond to that attribute name-value pair. The filter is always treated as an equals operation. The `filterName` must be an attribute that is indexed in the `FullTextIndex` object for the specified class.

Syntax	<code>textsearch classname string [filterName filterValue]</code>
Examples	<code>> textsearch Bundle manager type business</code>
Result	Returns a map of data values for each <code>Bundle</code> (role) of <code>type=business</code> that contains the string <code>manager</code> in any analyzed field. Analyzed fields in the <code>Bundle FullTextIndex</code> marked as <code>analyzed=true</code> .

Seldom Used Commands

These commands are used by developers for testing and are rarely used in a production environment.

Properties

The `properties` command displays system properties.

Syntax	<code>properties</code>
Examples	<code>> properties</code>
Result	Displays Java properties of the server on which IdentityIQ is installed

Time

The **time** command reports the duration of another command.

Syntax	<code>time <i>command</i></code>
Examples	<code>> time run "refresh risk scores"</code>
Result	Initiates the run command and then indicates how much time it took to run. Most useful for long-running commands

Xtimes

The **xtimes** command repeats a single command as many times as specified in the first argument. This command is used for performance testing purposes. Running a command numerous time provides a more accurate indication of how long a process takes than running it once.

Syntax	<code>xtimes <i>xcommand</i></code>
Examples	<code>> xtimes 3 run "refresh risk scores"</code>
Result	Runs the refresh risk scores task three times in a row

This command can be combined with the time command to report timing statistics on the performance test. By specifying this command first (for example, `xtimes 20 time run taskname`), the time taken for each command run is reported. By specifying the time command first (for example, `time xtimes 20 run taskname`), the total time for all of the sequential runs is reported.

About

The **about** command displays IdentityIQ's application configuration information.

Syntax	<code>about</code>
Examples	<code>> about</code>
Result	Lists application configuration specifics for the IdentityIQ instance (version, database, host, memory, etc.)

Threads

The **threads** command displays all active threads in the instance.

Syntax	<code>threads</code>
Examples	<code>> threads</code>
Result	Lists all active threads

LogConfig

The **logConfig** command reloads the log4j configuration into the instance.

Syntax	logConfig
Examples	> logConfig
Result	Reloads the log4j configuration from the log4j2.properties file

Summary

The **summary** command lists all classes and the count of objects of each class. Changes in these counts for some objects (for example, auditConfig) can indicate potential problems or areas of concern.

Syntax	summary
Examples	> summary
Result	Lists class name and count of objects for each class in the system

Rollback

The **rollback** command can undo a change to a role by restoring it from its BundleArchive object. BundleArchive objects are created when role archiving is enabled for IdentityIQ. Role archiving tracks changes made to a role by storing the pre-modification state in a BundleArchive object when the Bundle object is updated. This command only applies to the BundleArchive class.

Syntax	rollback <i>classname</i> < <i>objectname</i> or <i>id</i> >
Examples	> rollback BundleArchive "Contractor-IT"
Result	Restores the Contractor-IT role to the pre-modification state stored in its BundleArchive object

Rename

The **rename** command changes the name of an object from its existing name to the value specified by the *newname* parameter.

Syntax	rename <i>classname</i> < <i>objectname</i> or <i>ID</i> > <i>newname</i>
Examples	> rename application ADAM ADAM-Production
Result	Changes the name of the ADAM application to ADAM-Production

The object can be found using its old Name or its ID value, but in either case, the *newname* value is used to update the Name attribute for the object.

ExportJasper

The **exportJasper** command creates a JasperReport XML file from a JasperTemplate object in IdentityIQ. Jasper Report is a third party user interface for report writing. JasperReport XML is not compatible with IdentityIQ's XML so the JasperReport XML is wrapped in a JasperTemplate object when saved in IdentityIQ. The JasperTemplate must be exported to create a file that can be used directly with the Jasper user interface before it can be reformatted.

Syntax	<code>exportJasper filename <JasperTemplateName or ID></code>
Examples	<code>> exportJasper c:\data\AggResRpt.xml AggregationResults</code>
Result	Exports the Jasper XML from the AggregationResults JasperTemplate object into the file c:\data\AggResRpt.xml

The import command can be used to re-import a JasperReport object into the database. The import wraps the XML in a JasperTemplate.

Identities

The **identities** command lists the Name, Manager, Roles, and Links for each identity in the system. By default, this information prints to the console (stdout) and can be difficult to read due to screen wrapping. If the output is redirected to a file, it is printed in the file in an easy-to-read style.

Syntax	<code>identities</code>
Examples	<code>> identities</code> <code>> identities > identities.txt</code>
Result	The first example writes the Name, Manager, Roles, and Links for each identity in the system to the console (stdout). The second example redirects that information to the file identities.txt.

Snapshot

The **snapshot** command takes a snapshot of the named identity as it exists at that moment and archives it in the database as an IdentitySnapshot object. This object provides a historical record of the state of Identity objects at various points in time. Automatic snapshotting can be enabled and configured to create IdentitySnapshot objects at specified intervals or based on system activities (weekly, on aggregation change, etc.). The configuration of this feature can negatively impact system performance.

Syntax	<code>snapshot <identityname or ID></code>
Examples	<code>> snapshot Alan.Bradley</code>
Result	Creates an IdentitySnapshot object for the identity Alan.Bradley, capturing his Identity Attributes, Roles (Bundles), Entitlements Outside Roles, Links, and Scorecard information at that moment in time

Score

The **score** command refreshes the identity score for the named identity and updates that score in the database. Score updates are more commonly executed through the IdentityIQ user interface.

Syntax	<code>score <identityname or ID></code>
Examples	<code>> score Alan.Bradley</code>
Result	Recalculates the risk scores for Alan.Bradley and updates his Scorecard with the new risk scores

Tasks

The **tasks** command lists the Name, State, Next Execution, and Cron Strings for all currently scheduled tasks in the system.

Syntax	<code>tasks</code>
Examples	<code>> tasks</code>
Result	All currently scheduled tasks are written to the console (stdout)

TerminateOrphans

The **terminateOrphans** command sets the completion status of any open taskResult objects to Terminated. While tasks are running, their taskResults should be in a pending state, but occasionally task results can become orphaned and remain in this non-completed state when the task has finished (or has otherwise been terminated). This command can be used to clean up those orphaned taskResults but it must only be executed when there are no tasks running on the application server or the taskResults for actively running tasks are terminated along with any orphaned results.

This command requires no arguments for execution but an artificial argument `please` has been added to prevent accidentally running this command.

Syntax	<code>terminateOrphans please</code>
Examples	<code>> terminateOrphans please</code>
Result	Sets all open taskResults for the application server to the Terminated status

Certify

The **certify** command creates a manager or application certification. The certification is generated using the installation's default settings/parameters. This command is primarily used for testing purposes.

Syntax	<code>certify [managerName application]</code>
Examples	<code>> certify Catherine.Simmons</code>
Result	Generates a manager certification for manager Catherine Simmons

The command syntax help indicates that this command can generate an application owner certification when an application is specified as a command argument, but this feature has not been updated as the certification components of the product have changed over time. As a result, the application argument for this command is not currently usable.

CancelCertify

This command is not recommended. Use the delete command to remove certification objects.

The **cancelCertify** command can be used to delete a certification object from the system.

Syntax	<code>cancelCertify <certificationName or ID></code>
Examples	<code>> cancelCertify "Manager Access Review for William Moore"</code>
Result	Delete the named certification (the command fails if more than one certification object with the same name exists)

ArchiveCertification

The **archiveCertification** command archives the specified certification (creates a certificationArchive object) and deletes it as an active certification.

Syntax	<code>archiveCertification <certificationName or ID></code>
Examples	<code>> archiveCertification "Manager Access Review for William Moore"</code>
Result	Creates a certificationArchive object and delete the certification from the system

DecompressCertification

The **decompressCertification** command retrieves the named certificationArchive object and prints it to the console (stdout) in the Certification object's XML format.

Syntax	<code>decompressCertification <certificationArchiveName or ID></code>
Examples	<code>> decompressCertification "Manager Access Review for William Moore"</code>
Result	Prints the named certification archive to the console (stdout) in certification XML format

WorkItem

The **workItem** command displays certain details (Owner, Create Date, Expiration Date) for the specified workItem.

This command requires the workItem ID or name value as an input parameter. The workItem ID value (a long hexadecimal number) is obtained using the IdentityIQ console's list workItem command. The workItem Name is not the descriptive name for the workitem, it is a numeric value assigned when the workItem is created. The value is found in the XML representation of each workItem through the Debug pages.

Syntax	<code>workItem <workItemID or Name></code>
Examples	<code>> workItem 40288f0132b155ad0132b58a4e3f018e</code>
Result	Displays the Owner, Created Date, and Expiration Date for the specified workItem

Approve

The **approve** command sets the specified workItem to a Finished state (indicating it was approved), adds any specified completion comments to the workItem, and submits the workItem to the workflow to move it to the next appropriate stage.

This command requires the workItem ID or name value as an input parameter. You can obtain the workItem ID value (a long hexadecimal number) using the IdentityIQ console list workItem command. The workItem Name is not the descriptive name for the workitem, it is a numeric value assigned when the workItem is created. The value is found in the XML representation of each workItem through the Debug pages.

Syntax	<code>approve <workItemID or Name> [comments]</code>
Examples	<code>> approve 40288f0132b155ad0132b58a4e3f018e "Access approved"</code>
Result	Marks the specified workItem as approved, adds the comment "Access approved" to the workItem's completion comments, and submits the workItem for evaluation of the next appropriate step (another approval, provisioning, etc.)

Reject

The **reject** command sets the specified workItem to a Rejected state, adds any specified completion comments to the workItem, and submits the workItem to the workflow to move it to the next appropriate stage.

This command requires the workItem ID or name value as an input parameter. The workItem ID value (a long hexadecimal number) is obtained using the IdentityIQ console's list workItem command. The workItem Name is not the descriptive name for the workitem, it is a numeric value assigned when the workItem is created. The value is found in the XML representation of each workItem through the Debug pages.

Syntax	<code>reject <workItemID or name> [comments]</code>
Examples	<code>> reject 40288f0132b155ad0132b58a4e3f018e "Access conflicts with AP data entry entitlement"</code>
Result	Marks the specified workItem as rejected, adds the comment "Access conflicts with AP data entry entitlement" to the workItem's completion comments, and submits the workItem for evaluation of the next appropriate step (another approval, etc.)

Warp

The **warp** command parses an XML file to create an object and then displays the object's XML representation in the console (stdout). If it is not in valid form or its tags do not match the IdentityIQ DTD, a RuntimeException is printed to the console describing the error.

Syntax	<code>warp filename</code>
Examples	<code>> warp c:\data\newWorkflow.xml</code>
Result	Parses the XML in the file <code>c:\data\newWorkflow.xml</code> and displays the XML representation of the object in the console, or reports any errors to the console

Notify

The **notify** command sends an email message to the specified identity using the email template specified. This command does not accept any other parameters that can be passed to the template, so it can only be used for templates whose messages do not rely on variable substitutions to build the content. This command is most often used for testing purposes.

The `toAddress` argument can contain an identity name or ID or an email address. If it contains an identity name or ID, the email address is retrieved from the identity record.

Syntax	<code>notify <emailTemplateName or ID> toAddress</code>
Examples	<code>> notify Certification Alan.Bradley</code>
Result	Sends an email to Alan.Bradley's email address using the Certification email template

Authenticate

The **authenticate** command authenticates a username and password against the pass-through authentication source or the internal IdentityIQ records. No results are returned if the values are authenticated. If the password is incorrect or the user name cannot be found, an error message is displayed in the console (stdout).

Syntax	<code>authenticate usernamepassword</code>
Examples	<code>> authenticate Alan.Bradley s53n659#@5a!</code>
Result	Authenticates username Alan.Bradley and the provided password against the authentication source (pass-through or internal)

SimulateHistory

The **simulateHistory** command is used to generate a fake, randomly-generated group index or identity score history for one or more groups or identities. Used for generating test data in a development environment.

Syntax	<code>simulateHistory Identity Group <groupName or ID> <identityName or ID> all</code>
Examples	<code>> simulateHistory Identity all</code> <code>> simulateHistory Group Finance</code>
Result	First example generates fake risk scorecards for all identities in the system Second example generates fake groupIndex information for the Finance group

Search

The **search** command looks up an object based on specified criteria, similar to a simplified SQL/HQL interface. A single class name is specified with a list of the attributes to display from that class. Following the where keyword, search filters can be specified in name value sets. All filter values are used in a like comparison. The record is returned if the record's field value contains the specified value string.

Syntax	search <i>className</i> [<i>attributeName</i> ...] where [<i>filter</i> ...] filter: <i>attributeNamevalue</i>
Examples	> search identity name manager.name region where name kat
Result	Returns the name, manager's name, and region for all identities whose name contains the string kat. For example, records for Katherine.Jones, John.Kato, and Tammy.Erkatz are returned by this search

CertificationPhase

The **certificationPhase** command transitions the specified certification to the specified phase. This command fails if the certification is on or past the requested phase.

The certification is advanced to the next enabled phase after the requested phase if the specified phase is not enabled for the certification. For example, if a certification has neither a Challenge nor a Remediation phase enabled but the command requests that it be advanced to the Challenge phase, the certification is advanced to the End phase.

The certification is sequentially advanced through all enabled phases until it reaches or passes the requested phase. Any business logic that should occur during each phase transition (period enter rules, period end rules, etc.) is executed during the phase advancement.

Syntax	certificationPhase < <i>certificationName</i> or <i>ID</i> > [Challenge Remediation End]
Examples	> certificationPhase "Catherine Simmons Access Review" Challenge
Result	Advances the "Catherine Simmons Access Review" certification from its current phase (Active) to the Challenge phase. If this review is not configured for a Challenge phase, it is transitioned to the Remediation or End phase (depending on configuration).

Impact

The **impact** command reads an XML file containing a Bundle (role) object and performs role impact analysis for the role. The command parses the XML to its object form. Impact analysis is not performed if that object is not a Bundle.

Syntax	impact <i>filename</i>
Examples	> impact c:\data\ContractorRole.xml

Result	Performs role impact analysis for the Bundle object represented by the XML in c:\data\ContractorRole.xml
--------	--

Event

The **event** command schedules a workflow to run, passing in an Identity name as an argument. By default, the workflow is scheduled 1 second after the command is issued, but a delay can be specified in seconds as a command argument.

Syntax	<code>event <identityName or ID> <workflowName or ID> [seconds]</code>
Examples	<code>> event Catherine.Simmons "Identity Refresh" 60</code>
Result	Schedules an Identity Refresh workflow to run for Catherine.Simmons 60 seconds after the command is issued

ConnectorDebug

The **connectorDebug** command is used to test a connector or troubleshoot application aggregation issues. Its method parameters determine what is tested and how.

Syntax	<code>connectorDebug <applicationName or ID> <method> [methodArgs...]</code>
--------	--

The specific syntax for each of the "methods" is shown below.

Method	test
Purpose	Test whether a connection can be established with the application through its connector
Syntax	<code>connectorDebug <applicationName or ID> test</code>
Example	<code>> connectorDebug ADAM test</code>
Result	Returns "Test Succeeded" on success, reports an error in the console on failure.

Method	iterate
Purpose	Iterate through the application's account or group records
Syntax	<code>connectorDebug <applicationName or ID> iterate [account group (default = account)] [-q (for "quiet mode")]</code>
Example	<code>> connectorDebug ADAM iterate -q</code> <code>> connectorDebug ADAM iterate account</code>
Result	First example iterates all account records natively in the ADAM application and

returns only the count of iterated objects and how many milliseconds it took to run.

Second example iterates account records natively in the ADAM application and returns a ResourceObject representation of each account to the console.

Method	get
Purpose	Test whether a connection can be established with the application through its connector
Syntax	connectorDebug < <i>applicationName</i> or <i>ID</i> > get account group <i>nativeIdentity</i>
Example	> connectorDebug ADAM get account "CN=Willie.Gomez,DC=sailpoint,DC=com"
Result	Returns the XML representation of the ResourceObject for that nativeIdentity on the application

Method	auth
Purpose	Test pass-through authentication against the specified application (The featuresString in its application definition must contain AUTHENTICATION.)
Syntax	connectorDebug < <i>applicationName</i> or <i>ID</i> > auth <i>usernamepassword</i>
Example	> connectorDebug ADAM auth administrator Pa\$\$w0rd
Result	Returns "Authentication Successful" when user is authenticated or displays the exception message to the console if authentication fails

Encrypt

The **encrypt** command is used to encrypt a string. This command is generally only useful for test purposes. It can generate an encrypted password which can be passed in other console commands, for example, the **authenticate** command.

Syntax	encrypt <i>string</i>
Examples	> encrypt MyPa\$\$w0rd
Result	Returns the encrypted equivalent for the specified string

HQL

The **hql** command executes a search based on a Hibernate Query Language statement. The command syntax matches the sql command's syntax, but this command can select but not update data.

Syntax	<code>hql <i>hqlStatement</i> -f <i>inputFileName</i></code>
Examples	<pre>> hql "select name, manager.name from Identity" > c:\data\Identities.dat > hql -f c:\hql\SelectIdentities.hql</pre>
Result	<p>The first example executes the specified HQL select statement and writes the results to the file <code>c:\data\Identities.dat</code>.</p> <p>The second example reads the HQL from the file <code>c:\hql\SelectIdentities.hql</code>, prints the HQL to the console (stdout), and displays the query results to the console (stdout).</p>

Date

The **date** command shows the current date and time for the application server or the date and time value for a specified utime (universal time) value.

Syntax	<code>date [<i>utime</i>]</code>
Examples	<pre>> date > date 1338820492484</pre>
Result	<p>The first example displays the command syntax and the current date/time and current UTIME value.</p> <p>The second example returns the date/time value for the specified UTIME value.</p>

Shell

The **shell** command escapes out to the command line and runs the command specified. (This command does not work properly in a Windows environment but does work in UNIX.)

Syntax	<code>shell <i>commandLine</i></code>
Examples	<pre>> shell ls</pre>
Result	Lists the contents of the UNIX file system directory from which the console was run

Meter

The **meter** command toggles metering on or off. While metering is on, the console reports some timing statistics for each command executed. Meter information is displayed after the results of each command as it is executed.

Syntax	<code>meter</code>
Examples	<pre>> meter</pre>
Result	Toggles metering on and off. When turned on, all subsequently issued commands

report timing statistics.

Meter information displayed includes: number of calls, total number of milliseconds, maximum time for one call, minimum time for one call, and average time per call.

Compress

The **compress** command is designed to compress the contents of a file to a string that can be included within an XML element. It compresses the file and then encodes it to Base64 and writes that text to the specified output file. This resultant file can then be used in an XML element stored in the database. This has limited usefulness within IdentityIQ since no part of the application is designed to read these compressed strings, but custom rules can be used to process them as needed or they can simply be stored in the database to be retrieved and uncompressed for use by an external application at a later time.

Syntax	<code>compress <i>inputFilename</i><i>outputFilename</i></code>
Examples	<code>> compress file1.txt file2.txt</code>
Result	Compresses the contents of file1.txt, encodes that into Base64, and writes the resultant text string to file2.txt

Uncompress

The **uncompress** command functions in exactly the opposite way of the **compress** command, taking a compressed, Base64-encoded file and returning its uncompressed format.

Syntax	<code>uncompress <i>inputFilename</i><i>outputFilename</i></code>
Examples	<code>> uncompress file2.txt file3.txt</code>
Result	Reverses the compressing process to return the original, uncompressed version of the text, writing that to the file file3.txt

ClearEmailQueue

The **clearEmailQueue** command deletes all queued but unsent email messages from the IdentityIQ email queue. This includes any new messages that have not yet been sent and messages that have encountered problems that prevented successful delivery.

Syntax	<code>clearEmailQueue</code>
Examples	<code>> clearEmailQueue</code>
Result	Deletes all unsent emails from the email queue

ClearCache

The **clearCache** command removes objects from the Hibernate object cache. This can be used when debugging Hibernate issues.

Syntax	clearCache
Examples	> clearCache
Result	Clears the Hibernate object cache

Service

The **service** command provides information about the background services running in the console. The services include:

- Cache - periodically refreshes cached objects
- SMListener - listens for change events from PE2 change interceptors
- ResourceEvent - looks for change events added to a queue and processes them
- Heartbeat - maintains a Server object for each IdentityIQ instance and periodically updates it so you can tell if an instance is still running
- Task - the Quartz task scheduler
- Request - the IdentityIQ request processor - stopping the Request service also stops partitioned tasks

Syntax	service list start stop run
Examples	> service list
Result	Lists background services running in the console